AN ABSTRACT OF THE THESIS OF

Chengyun Chu for the degree of Master of Science in Computer Science

presented on June 1st, 1999.

Title: Test Case Prioritization

Redacted for privacy

Abstract approved: _____

Gregg Rothermel

Prioritization techniques are used to schedule test cases to execute in a specific order to maximize some objective function. There are a variety of possible objective functions, such as a function that measures how quickly faults can be detected within the testing process, or a function that measures how fast coverage of the program can be increased. In this paper, we describe several test case prioritization techniques, and empirical studies performed to investigate their relative abilities to improve how quickly faults can be detected by test suites. An improved rate of fault detection during regression testing can provide faster feedback about a system under regression test and let debuggers begin their work earlier than might otherwise be possible. The results of our studies indicate that test case prioritization techniques can substantially improve the rate of fault detection of test suites. The results also provide insights into the tradeoffs among various prioritization techniques.

Test Case Prioritization

by

Chengyun Chu

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 1st, 1999
Commencement June 2000

Master of Science thesis of Chengyun Chu presented on June 1st, 1999

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Chair of the Department of Computer Science

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Chengyun Chu, Author

ACKNOWLEDGMENTS

I would especially like here to express my appreciation to my major professor, Dr. Gregg Rothermel, for all the time and energy he has spent to guide me to finish this thesis. I am lucky to work with such a good professor.

Thanks also to Dr. Tim Budd, for serving as my minor professor, and to Dr. Margaret Burnett, for being on my commitee.

Thanks to everybody in the Aristotle research group, particularly to Dr. Untch, who was involved in the statistical analysis of empirical studies of prioritization techniques, Jim Law, to whom I always asked questions about the Aristotle system, Jeff Ronne, who updated the test matrix driver and sensitivity matrix driver used in the experiment, and Qiang Liu, who contributed to early discussion of the work. Thanks also to all the other people who gave me help in this thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

TEST CASE PRIORITIZATION

Chapter 1

INTRODUCTION

Software plays an increasingly important role in our society. As software systems become larger and more complex, the importance of software testing, which tries to reveal potential faults and ensures system qualities, also increases.

No matter how well a software system has been tested, it will inevitably be modified to fix faults or implement new functionality required by users. Once a software system is modified, regression testing is performed to provide confidence that the quality of the system is also maintained after the changes. More precisely, regression testing is the testing "performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program" [14].

Regression testing is expensive. It has been estimated that up to two thirds of the overall budget of software production is used for software testing and maintenance activities [13], and among these tasks, regression testing may account for as much as 50% of the costs [9]. For mission-critical software, such as air traffic control systems, even more effort for regression testing may be needed.

The fundamental difference between regression testing and other testing activities is that during regression testing, a test suite developed for the previous

version of the software is available for reuse. Sometimes, however, simply running all test cases in that test suite can require a great deal of effort. For instance, it is reported that for one software system that contains about 20,000 lines of code, it would take seven weeks to run the entire test suite [11]. Thus, we would like to find some way to reuse test suites more efficiently.

People have looked at a variety of methods for reducing the cost of regression testing, including test selection and minimization techniques. Regression test selection techniques reduce the cost of regression testing by selecting an appropriate subset of the existing test suite based on information about the code of the program and the modified version or about program specifications [15]. Minimization techniques lower costs by reducing a test suite to a minimal subset that maintains equivalent coverage of the original test suite with respect to a particular test adequacy criterion [16].

These methods, however, may have potential problems. For example, although some researchers argue that there is little or no loss in the ability of a minimized test subset to reveal faults [20], another recent empirical study reveals that the fault detection capabilities of test suites can be severely compromised by minimization [16]. Also, although there are safe regression test selection techniques (e.g. [1, 2, 15, 18]) that assure the selected subset has the same fault detection capability as the original suite, the conditions under which safety can be achieved may not always be able to hold [14, 15].

Test case prioritization techniques provide another way to improve the cost-effectiveness of reusing test suites. Based on information associated with a test suite, such as coverage information or information estimating test cases' potential ability to reveal faults, test case prioritization techniques schedule test cases to execute in a specific order to maximize some objective function.

For example, testers may seek to detect faults as quickly as possible, or increase coverage of the program at the fastest rate possible.

When the size of the test suite is small and all test cases can be run quickly, simply running all test cases in any order may be more cost-effective than performing test case prioritization and then running the test cases. However, when the time required to run the whole test suite is long, test case prioritization techniques may bring benefits by scheduling the most important test cases to run earlier. For example, suppose there are time and resource constraints that let only a part of the test suite be run each day. In this case, it may be better to use test case prioritization techniques to cause the most important test cases for achieving the tester's objectives to execute in the first day, rather than running test cases in a random order.

Test case prioritization techniques can also be used in combination with test selection and minimization methods. For example, testers may use selection or minimization first to produce a smaller test suite, and then use prioritization methods to schedule a specific order for executing the test cases in this smaller suite. One study [21] suggests some techniques for combining prioritization, minimization and test selection methods; however, the concept of prioritization used in that study is somewhat different from the one that we consider.

In this paper, we present several test case prioritization techniques. The prioritization techniques fall into two categories: techniques that perform prioritization based on coverage information and techniques that perform prioritization based on fault exposing potential. We also present random prioritization and optimal prioritization techniques, that work as "benchmarks" to let us evaluate the performance of other test case prioritization techniques.

We describe our empirical studies in which we investigate these prioritization techniques' ability to improve the rate of fault detection of test suites. The rate of fault detection measures how quickly faults are detected in the testing process. The higher the rate of fault detection, the earlier the feedback on faults is returned to the developer, and thus, the earlier the developer can begin debugging the code and fixing the faults. The results of our empirical study strongly indicate that test case prioritization techniques can significantly improve the rate of fault detection. Meanwhile, our results show the various effects of these prioritization techniques and highlight the tradeoffs between them.

In the next chapter, we present background information on test case prioritization techniques. Chapter 3 defines the test case prioritization problem and describes several prioritization techniques. Chapter 4 discusses the implementation of our prioritization techniques, and then presents the design and results of our empirical studies, which consist of three experiments. In Chapter 5, conclusions are drawn and future work is discussed.

Chapter 2

BACKGROUND

2.1 Software testing

Software testing is the process of executing a program with input data and determing whether the program's output matches its specification or not. To introduce the terminology of software testing in a precise manner, we use the following notation:

Let P be a program, let D be the input domain of P, and let F be the hypothetical correct version of P (F is called an oracle and may be obtained from a specification). A test case t is an element of D. A test suite T is a finite set of test cases. If for a test case t, $P(t) \neq F(t)$, we say that a failure is demonstrated.

If there are no faults revealed by a given test suite T, we cannot guarantee that program P is 100% correct since in general, the input domain D is infinite. However, if test suite T is designed systematically, we should have greater confidence in the quality of P.

After software is modified, regression testing should be performed to test not only that the changes are correct, but that unchanged parts of the program have not been corrupted by the changes as well. During regression testing, we have a program P, its modified version P', and a test suite T created to test P.

In the following sections, we discuss two categories of information used as a basis for test case prioritization techniques to decide the order of execution of test cases. One category is coverage information used in coverage-based prioritization techniques, the other is fault-exposing-potential (FEP) information used to estimate a test case's potential ability to reveal faults in fault-exposing-potential (FEP) prioritization techniques.

2.2 Coverage information

2.2.1 Control flow graphs

A *control flow graph* (CFG) represents the control flow structure of a procedure. A node in a CFG represents a simple or conditional statement, while an edge represents control flow between statements.

Figure 2.1 shows procedure **avg** and its control flow graph. Procedure **avg** takes an integer array *a* and an integer variable *count* as its input parameters. Array *a* can hold up to 10 integers and variable *count* records the number of integers actually stored in *a*. Procedure **avg** returns the average of integers in *a*. If *count* is out of range, it returns -1. In the CFG, nodes *entry* and *exit* represents entry to and exit from **avg**, respectively. Ellipses, acting as statement nodes, represent simple statements. Rectangles, acting as predicate nodes, represent conditional statements. Each predicate node has two out edges labeled "T" and "F". These two edges represent control flow paths taken when the predicate evaluates to the value of the edge label. Nodes are labeled with their corresponding statement numbers.

```
avg(a: array[1..10] of int,count:int,result:int)
    int i, sum
    begin
1.      result = -1;
2.      if(count >=0 AND count <=10)
3.          sum = 0
4.          i = count
5.          while(i > 0) do
6.              sum = sum + a[i]
7.              i = i - 1
            end
8.          result = sum/count
        end
    end
```

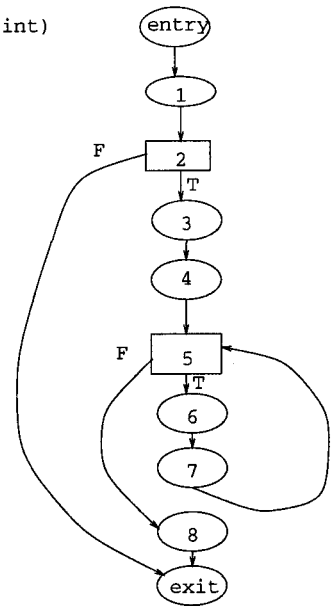

FIGURE 2.1: Control flow graph of procedure avg.

2.2.2 Node coverage, edge coverage, test history

Coverage information for test case t on program P is acquired by executing an instrumented version of program P with test t. For node coverage[1], an instrumented version of P records the node trace for t, that consists of the nodes covered during the execution of P with t. For branch coverage, branches covered by test t are recorded. The branch trace information could be used to generate the edge coverage information. An edge (n_1, n_2) in a control flow graph is covered by test t if and only if, when P is executed with t, the statements

[1] An equivalent term for node coverage is "statement coverage", because a node in a CFG represents a simple or conditional statement.

associated with n_1 and n_2 are executed sequentially at least once during the execution.

Table 2.1 illustrates node coverage information for a test case on **avg**. Table 2.2 illustrates edge coverage information for two test cases on **avg**.

Test	Inputs	Node coverage
t1	{a=10,2,4,6; count=4}	{E,1,2,3,4,5,6,7,8,X}

TABLE 2.1: Node coverage for test t1 on procedure **avg**.

Test	Inputs	Edge coverage
t1	{a=10,2,4,6; count = 4}	(E,1),(1,2),(2,3),(3,4),(4,5), (5,6),(6,7),(7,5),(5,8),(8,X)
t2	{a=10,2,4,6; count=-2}	(E,1), (1,2), (2,X)

TABLE 2.2: Edge coverage for test t1, t2 on procedure **avg**.

For a given test suite T for program P, with CFG G, its *edge test history* is built by collecting edge coverage information for each test case t in T and representing it such that for each edge in G, the test history records the tests

that cover this edge. Table 2.3 gives an example of edge test history. A similar definition applies for a node (statement) test history.

Edge	TestsOnEdge
(E,1)	t1, t2
(1,2)	t1, t2
(2,3)	t1
(2,X)	t2
(3,4)	t1
(4,5)	t1
(5,6)	t1
(6,7)	t1
(7,5)	t1
(5,8)	t1
(8,X)	t1

TABLE 2.3: Edge test history for procedure **avg** of test suite $\{t1, t2\}$

As we shall discuss in Chapter 3, our coverage-based test case prioritization techniques prioritize test suite T for program P based on the control flow graph G of P and the test history, either node or edge, of T for P.

2.3 FEP information

The idea of *fault-exposing-potential* (FEP) is initiated by Voas' paper on code sensitivity analysis [17]. Voas presents a technique called propagation, infection and execution (PIE) analysis, used to evaluate the sensitivity of a specific statement in a program. Voas defines a statement's sensitivity as a prediction of the minimal probability that a fault in this statement will result in a program failure under a particular input distribution.

To compute the sensitivity of statement s of program P under a particular input distribution $\mathbf{D}$, PIE analysis evaluates the following three probabilities:

1. Execution probability (PE) is the probability that statement s could be reached. The procedure to estimate PE is: randomly select n input data sets according to distribution $\mathbf{D}$, and execute the instrumented version of program P with these n input data sets. If m of these data sets hit statement s, the PE of s is $\frac{m}{n}$.

2. Infection probability (PI) is the probability that a change in source statement s could result in the data state that holds after s executes being infected (altered). To estimate PI, one approach is to generate code mutants. A mutant is a copy of the original program into which small modifications have been inserted. For example, an addition operator may be replaced by a subtraction operator. These code mutants for statement s are then executed with randomly selected data states to estimate whether the states after statement s are changed.

3. Propagation probability (PP) is the probability that an infected data state after statement s could result in the output of P being changed. To

estimate PP, we randomly select n data states after statement s, and execute the rest of program P with these n data states. If m of them alter the output, the PP of s is $\frac{m}{n}$.

Once PE, PI, PP have been estimated, θ, the sensitivity of statement s, is calculated according to the following formula:

$$\theta = PE \cdot \sigma\,(PI,\ PP)$$

where $\sigma\,(a,\ b) = a - (1 - b)$ if $a - (1 - b) > 0$; otherwise, $\sigma\,(a,\ b) = 0$.

PIE sensitivity analysis provides a primitive approach for estimating the fault exposing potential of test cases. For example, if under a particular input distribution $\mathbf{D}$, the sensitivity of statement s of program P is 1.0, then it is predicted that each input in $\mathbf{D}$ will result in a failure of P if statement s contains a fault. Therefore, if test case t is selected from $\mathbf{D}$, its probability of revealing a fault hiding in statement s is 1.0.

However, there are some factors in PIE analysis that prevent us from directly using it in our test case prioritization techniques. In the description of FEP prioritization methods, we will further explore this idea and introduce the way we estimate the fault exposing potential of test cases.

2.4 Related work

A recent study by Wong, Horgan, London and Agrawal [21] suggests prioritizing test cases in order of increasing cost per additional coverage. However, the authors use prioritization to reduce the size of the test suite, rather than schedule an order of execution of the test suite. Their procedure first sorts test cases of a test suite selected by a safe regression test selection technique from

the original test suite for the program, and then selects the top n test cases for revalidation of the modified version. The authors do not specify a mechanism for prioritizing the remaining tests after full coverage has been achieved. Such a mechanism is necessary if we wish to prioritize all tests – even those remaining after full coverage is achieved. The authors describe an empirical study in which they applied their technique to a program of 5000 lines of code, evaluated the performance of their technique in reducing test suites against ten faulty versions of that program, and concluded that the technique was cost-effective.

In our work, we further investigate coverage-based prioritization techniques, but we examine a wider range of techniques. We also focus on the effects of prioritization on maximizing the rate of fault detection of test suites.

Chapter 3

TEST CASE PRIORITIZATION: PROBLEMS AND TECHNIQUES

3.1 The test case prioritization problem

Test case prioritization techniques schedule test cases for execution in an order that maximizes some objective function. The purpose of prioritization is to increase the likelihood that this objective function would be better met if the test cases used for regression testing are executed in the given order than if the test cases were executed in an ad-hoc order.

We define the test case prioritization problem formally as follows:

Test Case Prioritization Problem:

> *Given*: T, a test suite, PT, the set of permutations of T, and f, a function from PT to the real numbers.
>
> *Problem*: Find $T' \in PT$ such that $(\forall T'')\ (T'' \in PT)\ (T'' \neq T')$ $[f(T') \geq f(T'')]$.

In this definition, PT represents the set of all possible prioritizations (execution orders) of T, and the objective function f calculates an *award value* for each ordering of PT that measures its score to satisfy the objective function f. (For simplicity, we assume that higher award values are preferable to lower ones.)

Since in general, a test suite is defined as a finite set of test cases, and thus, mathematically, is unordered, we would like to emphasize here that in the test case prioritization problem, a test suite is defined as a *list* of test cases. The sequence of test cases in the test suite (list) is the order of execution of test cases.

In some test suites, there are dependencies between test cases: some tests can be run only after the execution of some other test cases. To simplify the problem, we assume that test cases are independent of each other.

There are various goals that can be addressed in test case prioritization. Possible goals include the following:

1. Testers may wish to increase the rate of fault detection - that is, the likelihood of revealing faults earlier in the testing process than would be possible with an ad-hoc test ordering.

2. Testers may wish to increase the rate of detection of high-risk faults - that is, locate such faults earlier in the testing process than would be possible with an ad-hoc test ordering.

3. Testers may wish to increase the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process than would be possible with an ad-hoc test ordering.

4. Testers may wish to increase the coverage of coverable code of the program under test at a faster rate than would be possible with an ad-hoc test ordering.

5. Testers may wish to increase the confidence in the reliability of the system under test at a faster rate than would be possible with an ad-hoc test ordering.

For each goal, an objective function needs to be defined to measure a prioritization technique's ability to meet that goal. Depending upon which goal and objective function are selected, the test case prioritization problem may be intractable; an efficient solution to the problem would provide an efficient solution to the knapsack problem [5]. Therefore, heuristics may be required for test case prioritization. A purpose of this work is to investigate the usage of several such heuristics for a specific objective function.

3.2 Prioritization techniques

For a given goal, various prioritization techniques can be applied to a test suite attempting to schedule an order of test case execution that better meets that goal. For instance, if the goal is to increase the rate of fault detection, we may prioritize test cases based on their historically determined capabilities of revealing faults, or we may prioritize test cases in terms of their increasing cost-per-coverage of code components, assuming that there is some correlation between fault detection effectiveness and code coverage. The selection of a prioritization technique depends upon particular circumstances. But in any case, the intention behind the choice is the same: to increase the likelihood that the prioritized test suite can meet the objective function more closely than would an ad-hoc test case ordering.

In this work, we focus on meeting the first goal listed in Section 3.1: increasing the likelihood of revealing faults earlier in the testing process. This goal is

achieved by improving a test suite's *rate of fault detection*, a measure of how quickly faults are detected in the testing process. A precise measurement for this is described in Section 4.1.2. The benefit of meeting this goal is apparent: an improved rate of fault detection means that feedback on faults is returned earlier to the developer, and thus, the developer can begin debugging the code and fixing the faults earlier than might otherwise be possible.

We investigate nine different test case prioritization techniques. Table 3.1 lists these techniques.

Code	*Mnemonic*	*Description*
$\mathcal{M}_1$	untreated	no prioritization (control)
$\mathcal{M}_2$	random	randomized ordering
$\mathcal{M}_3$	greedy-optimal	ordered to optimize rate of fault detection
$\mathcal{M}_4$	edge-total	in order of coverage of edges
$\mathcal{M}_5$	edge-addtl	in order of coverage of edges not yet covered
$\mathcal{M}_6$	FEP-total	in order of total probability of exposing faults
$\mathcal{M}_7$	FEP-addtl	in order of total probability of exposing faults, adjusted to consider effects of previous tests
$\mathcal{M}_8$	stmt-total	in order of coverage of statements
$\mathcal{M}_9$	stmt-addtl	in order of coverage of statements not yet covered

TABLE 3.1: A catalog of prioritization techniques.

3.2.1 No prioritization

To facilitate our empirical study, we consider one prioritization "technique" that is in fact the application of no technique. We keep the original test suites "untreated" and measure their rates of fault detection. Note, however, that the value of an untreated test suite in meeting the objective function may depend upon the way in which it is initially constructed.

3.2.2 Random prioritization

A random prioritization technique works as a "benchmark" to let us evaluate the performance of other test case prioritization techniques. It provides a basis for measuring how well non-random prioritization techniques can improve the rate of fault detection of a test suite, compared with simply executing test cases in a random order.

We describe our random prioritization algorithm in Figure 3.1.

The time complexity of random prioritization is analyzed as follows. Assume the size of the input test suite T is n. If the test suite is stored as an array, the time required to access and append one test case is $O(1)$, and the time required to delete one test case is $O(n)$. Therefore, the overall time complexity of random prioritization is $O(n^2)$.

3.2.3 Greedy-optimal prioritization

Greedy-optimal prioritization is also used in our empirical study as a benchmark for judging other prioritization techniques. As we shall describe in Chapter 4, to measure the effect of prioritization techniques on rate of fault detection, we use programs that contain known faults as experimental subjects in our empirical

Algorithm RandomPrioritization

Input: Test suite T

Output: Prioritized test suite T'

1. **begin**
2. set T' empty
3. **while** T is not empty do
4. randomly select a test case t from T
5. append t to T'
6. remove t from T
7. **endwhile**
8. **end**

FIGURE 3.1: Algorithm for random prioritization

studies. For a given test suite, we can determine which faults are revealed by the test cases in that suite, and thus, we can determine an optimal execution ordering of those test cases to maximize the test suite's rate of fault detection. This prioritization technique is not a practical technique, because it requires prior knowledge of which tests will expose which faults. However, by using it in our study, we gain insight into the success of other prioritization methods.

We describe the overall structure of the algorithm for the greedy-optimal prioritization in Figure 3.2. In this algorithm, fault-detection matrix M_{fd} is the data structure that stores the information about which tests expose which faults. Vector V_{fd} records the faults detected by selected test cases.

The time complexity of optimal prioritization is analyzed as follows. Assume the size of the input test suite T is n, and the number of faults is m. The while

Algorithm OptimalPrioritization

Input: Test suite T, fault-detection matrix M_{fd}

Output: Prioritized test suite T'

```
1.  begin
2.      set T' empty
3.      initialize vector V_fd to be empty
4.      while T is not empty do
5.          for each test case t in T
6.              compute the number of additional faults t detects for V_fd
7.          endfor
8.          select test case t that detects the most additional faults for V_fd
9.          append t to T'
10.         add faults detected by t to V_fd
11.         if V_fd is full then
12.             reset V_fd to be empty
13.         endif
14.         remove t from T
15.     endwhile
16. end
```

FIGURE 3.2: Algorithm for optimal prioritization

loop between lines 4 and 15 executes at most n times, and the loop between lines 5 and 7 executes at most n times. For line 6, the time required to calculate the number of additional faults detected by one test case is $O(m)$. Therefore, the overall time complexity of optimal prioritization is $O(n^2 \cdot m)$.

The algorithm of Figure 3.2 is a "greedy" algorithm. It cannot guarantee that its output is optimal in all cases. To see this, assume program P has four faults, and three test cases. Its fault-detection matrix is shown in Table 3.2. The greedy-optimal prioritization technique may select t_1 first, and t_2 second, and then the prioritized test suite it outputs is $\{t_1, t_2, t_3\}$. However, the real "optimal" execution ordering is $\{t_2, t_3, t_1\}$. Despite this fact, we believe that the greedy-optimal algorithm provides a useful benchmark against which to measure other techniques.

Test Case	Fault			
	1	2	3	4
t_1	X	X		
t_2	X			X
t_3		X	X	

TABLE 3.2: Fault-detection matrix of program P

3.2.4 Total edge coverage prioritization

By executing the instrumented version of program P with test case t, we can collect branch trace information for that test, and then generate its edge coverage information from which we can determine the number of edges in program P that were exercised by test case t. Total edge coverage prioritization prioritizes

test cases according to the total number of edges they cover simply by sorting them in order of total edge coverage achieved.

We describe the algorithm for total edge coverage prioritization in Figure 3.3. In this algorithm, edge trace history TH_{edge} is the data structure that records the information on which tests cover which edges in program P.

Algorithm TotalEdgeCoveragePrioritization
Input: Test suite T, edge trace history TH_{edge}
Output: Prioritized test suite T'
1. **begin**
2. set T' empty
3. **for** each test case t in T
4. calculate the number of edges t covers in P based on TH_{edge}
5. **endfor**
6. sort T in descending order based on total edge coverage
7. let T' be T
8. **end**

FIGURE 3.3: Algorithm for total edge coverage prioritization

The time complexity of total edge coverage prioritization is analyzed as follows. Assume the size of test suite T is n, and the number of edges in program P is m. The time required to calculate the total number of covered edges for one test case is $O(m \cdot QueryTime)$, where $QueryTime$ is the time required to query TH_{edge} to decide whether an edge is covered by a specific test

case. $QueryTime$ is a constant c once TH_{edge} is loaded. Thus, if T contains n test cases, the whole time required to calculate coverage information is $O(n \cdot m)$. The worst case time for sorting the test suite is $O(n \log n)$ using an appropriate algorithm. Therefore, the overall time complexity is $O(n \cdot m + n \log n)$. In general $m >> n$, in which case the time complexity of total edge coverage prioritization is $O(n \cdot m)$.

3.2.5 Additional edge coverage prioritization

As described above, total edge coverage prioritization sorts test cases in the order of total coverage achieved: to select a new test case, we choose the test that gives the maximal edge coverage among the unselected test cases, without considering information about edges covered by test cases already selected. However, having executed several test cases and covered certain edges, more may be gained by selecting tests to cover edges that have not yet been covered.

Additional edge coverage prioritization first selects a test case that yields the greatest edge coverage, then selects subsequent tests based on their additional edge coverage, which is the coverage of edges not yet covered by test cases already selected. Having begun to order tests in this way, we may have a problem: once all edges covered by at least one test case have been covered by selected test cases, no additional edge coverage can be gained by the remaining tests. We could order these tests next using any prioritization technique. In this work, we proceed by assuming that all edges are not yet covered, and apply additional edge coverage prioritization to the remaining tests again. The above process is repeated until all test cases have been prioritized.

We describe the algorithm for additional edge coverage prioritization in Figure 3.4. In this algorithm, TH_{edge} is the edge trace history of test suite T for program P. Vector $V_{coverage}$ records the edges covered by selected test cases.

Algorithm AdditionalEdgeCoveragePrioritization
Input: Test suite T, edge trace history TH_{edge}
Output: Prioritized test suite T'

```
1.  begin
2.      set T' empty
3.      initialize vector V_coverage to be empty
4.      while T is not empty do
5.          for each test case t in T
6.            compute additional edge coverage of t with respect to V_coverage
7.          endfor
8.          select test t that adds the most additional coverage for V_coverage
9.          append t to T'
10.         add the edges covered by t to V_coverage
11.         if V_coverage reaches full coverage then
12.             reset V_coverage to be empty
13.         endif
14.         remove t from T
15.     endwhile
16. end
```

FIGURE 3.4: Algorithm for additional edge coverage prioritization

Note that, in the additional edge coverage prioritization algorithm, when $V_{coverage}$ reaches full coverage, this means that all edges covered by T have been covered, not that all edges of program P have been covered, because there is no guarantee that T covers every edge of program P.

The time complexity of additional edge coverage prioritization is analyzed as follows. Assume the size of test suite T is n, and the number of edges in program P is m. The time required to calculate the additional edge coverage for one test case is $O(m)$. Thus the time required to select the test case which yields the most additional coverage from n test cases is $O(n \cdot m)$. This cost dominates the cost of the `while` loop between lines 4 and 15. The whole loop itself executes n times. Therefore, the overall time complexity of the algorithm is $O(n^2 \cdot m)$.

The additional edge coverage prioritization algorithm, like the greedy-optimal prioritization algorithm, is also a "greedy" algorithm. It can not guarantee that the edge coverage of program P will increase at the fastest rate by executing the prioritized test suite. The example used in relation to the greedy-optimal prioritization algorithm can also be used here to illustrate this problem, if we replace the word "fault" with "edge" in Table 3.2.

3.2.6 Total fault-exposing-potential (FEP) prioritization

Edge-coverage-based prioritization considers only whether an edge or statement has been covered by a test case. However, the ability of a fault to be exposed by a test case depends not only on whether the test case executes the faulty statement, but also on the probability that a fault in that statement will result in a program failure for that test case [17]. Although the exact determination

of this probability is infeasible in practice, we would like to investigate methods for approximating this probability, and determine whether the use of such an approximation could yield a prioritization technique superior in terms of rate of fault detection than techniques based on simple code coverage.

The idea of fault-exposing-potential (FEP) is initiated by Voas' paper on PIE analysis. Referring back to Section 2.3, PIE analysis evaluates the following three probabilities: execution probability (PE), infection probability (PI) and propagation probability (PP). The formula to calculate the sensitivity θ of statement s is:

$$\theta = PE \cdot \sigma(PI, PP)$$

where $\sigma(a, b) = a - (1 - b)$ if $a - (1 - b) > 0$; otherwise, $\sigma(a, b) = 0$.

Although PIE sensitivity analysis provides an approach for estimating the fault exposing potential of test cases, there are three problems that prevent us from directly applying it in our test case prioritization techniques for regression testing:

1. The purpose of PIE analysis is to predict the probability that a fault in a statement will result in a program failure under a particular *input distribution*, not for a particular test case. For a test case selected under an input distribution, PIE analysis can not decide whether this test will cover the statement, so probability PE is introduced. However, for test case prioritization techniques, by executing the instrumented program and collecting trace information, we can determine whether a test covers a statement. This coverage information can make a significant difference in estimating fault exposing probabilities, especially in the situation that we know a test goes through a statement which is hard to reach. Let us consider the following example. Suppose for statement s, the probabilities

of PI and PE are both 1.0, and the execution probability PE is 0.0001. According to Voas's formula, if s contains a fault, the predicted probability of exposing this fault is 0.0001, when we execute a test case t. However, based on the knowledge that t covers s, we can guarantee that the fault in s will be revealed by execution of t.

2. Estimating infection probability and propagation probability is difficult. There are obstacles in creating the sets of data states needed by infection and propagation analysis [17]. Thus, PIE analysis has only been partially automated and is difficult to apply to large software systems.

3. Function $\sigma\,(PI,\ PP)$ in formula $\theta = PE \cdot \sigma\,(PI,\ PP)$ may be too conservative for approximating sensitivities of statements. Since PIE analysis prefers an underestimated sensitivity to an overestimated sensitivity, the σ function is designed to reflect the worst case that the set of data state errors that produce the infection estimate is exactly the set of data state errors that does not propagate to the output, although this case is unlikely to occur. However, this preference may be unnecessary for test case prioritization techniques. If a large proportion of the fault exposing probabilities are underestimated, the capability of fault exposing information for guiding prioritization techniques in selection of test cases may be compromised.

To avoid these problems, we can use different approaches to estimate the fault exposing probability of a statement. To address the first problem, we can consider the fault exposing probability of a statement for a particular test case, instead of under a particular input distribution. Therefore, we can utilize the

coverage information about which tests cover which statements. To address the second problem, we can estimate the probability that a fault in a statement will be detected when executing a test that covers this statement. This probability, which can be called detection probability (PD), is a combination of infection probability and propagation probability. To estimate PD for a statement, we can apply mutation analysis to this statement and use tests that cover this statement. In this way, we avoid creating the sets of data states needed by infection and propagation analysis. Finally, to address the third problem, we can replace the σ function with other functions, such as $PI \cdot PP$, to avoid the preference for underestimated sensitivities.

In our empirical study, we obtain our approximation of the fault-exposing potential (FEP) by using mutation analysis. Given program P and test suite T, the FEP of a test case $t \in T$ for a statement s in P is the probability that a fault in statement s will be revealed by executing t. Compared with PIE analysis, probability PE is now diminished, for we now consider the fault exposing potential of statement s regarding a particular test case t and we know whether t covers s or not. FEP is the combination of infection probability and propagation probability. In fact, if we assume that the distribution of the set of data state errors that produce the infection estimate is independent of the distribution of the set of date state errors that propagate to the output, $PI \cdot PP$ can be an estimation of FEP.

Our formula to approximate the FEP of test case t for statement s is:

$FEP(t, s) = \frac{number\ of\ mutants\ of\ s\ exposed\ by\ t}{number\ of\ mutants\ of\ s}$

If test case t does not cover s, $FEP(t, s) = 0$.

The total FEP of test case t is defined as the sum of the $FEP(t, s)$ for all statements s covered by t. That is,

$$FEP(t) = FEP(t, s_1) + FEP(t, s_2) + \cdots + FEP(t, s_k)$$

where s_1, s_2, $\cdots$, s_k are statements covered by t.

The total FEP prioritization technique prioritizes tests according to their total FEP values. It sorts the tests in descending order of their FEP values.

We describe the algorithm for total FEP prioritization in Figure 3.5. It is very similar to total edge coverage prioritization. In this algorithm, TH_{edge} is the edge trace history. M_{FEP} is the FEP matrix which records the values of $FEP(t, s)$ for each test case and each statement.

Algorithm Total_FEP_Prioritization

Input: Test suite T, edge trace history TH_{edge}, FEP matrix M_{FEP}

Output: Prioritized test suite T'

1. **begin**
2. set T' empty
3. **for** each test case t in T
4. calculate $FEP(t)$ based on TH_{edge} and M_{FEP}
5. **endfor**
6. sort T in descending order based on $FEP(t)$
7. let T' be T
8. **end**

FIGURE 3.5: Algorithm for total FEP prioritization

The time complexity of total FEP prioritization is analyzed as follows. Assume the size of test suite T is n, and the number of statements in program P

is m. The time required to calculate $FEP(t)$ for a test t is $O(m \cdot QueryTime)$, where $QueryTime$ is the time required to query M_{FEP} to get $FEP(t, s)$. $QueryTime$ is a constant c once M_{FEP} is loaded. Thus, if T contains n test cases, the whole time required to calculate FEP information is $O(n \cdot m)$. The worst case time for sorting the test suite is $O(n \log n)$ using an appropriate algorithm. Therefore, the overall time complexity is $O(n \cdot m + n \log n)$. In general $m >> n$, in which case the time complexity of total FEP prioritization is $O(n \cdot m)$.

To execute total FEP prioritization algorithm, we need the FEP matrix, which is conceivably expensive to build since it needs data from mutation analysis. However, if such FEP techniques show promise, we might look for more cost-effective methods to approximate fault-exposing potential in further studies.

3.2.7 Additional fault-exposing-potential (FEP) prioritization

Similar to the extensions made to total edge coverage prioritization to obtain additional edge coverage prioritization, we extend total FEP prioritization to create additional FEP prioritization. One important concept we need to introduce for the additional FEP prioritization technique is the confidence of a statement. The confidence of a statement s is the probability that statement s is correct. If we execute a test case t which covers s and no fault is revealed, our confidence of s should be increased.

Assume that before execution of t, the confidence of statement s is $C(s)$, and the fault exposing probability of t for s is $FEP(t, s)$, then, after execution of t and if no failure results, the new confidence of s should be:

$C'(s) = 1 - (1 - C(s)) \cdot (1 - FEP(t, s))$

Simplifing the above formula, we have

$C'(s) = C(s) + (1 - C(s)) \cdot FEP(t, s)$

So, the additional confidence of statement s we gain by executing test t is:

$C_{addi}(s) = C'(s) - C(s) = (1 - C(s)) \cdot FEP(t, s)$

From the above formulas, we can see two properties of $C(s)$:

1. $C(s)$ is between 0 and 1.

2. $C(s)$ is increasing as tests are executed.

Now, we define $C_{addi}(t)$, the additional confidence gained from test case t, as the sum of $C_{addi}(s)$ for all statements s covered by t. If $s_1, s_2, \cdots, s_k$ are statements covered by t, then

$C_{addi}(t) = C_{addi}(s_1) + C_{addi}(s_2) + \cdots + C_{addi}(s_k)$

Additional FEP prioritization iteratively selects a test case t that gives the greatest additional confidence according to the current confidence of statements, then updates the confidence of statements covered by t and adjusts the additional confidence of all remaining test cases based on the updated confidence of statements, and then repeats this process until all tests have been prioritized.

We describe the algorithm for additional FEP prioritization in Figure 3.6. It is similar to the algorithm for additional edge coverage prioritization. In this algorithm, TH_{edge} is the edge trace history. M_{FEP} is the FEP matrix.

The time complexity of additional FEP prioritization is analyzed as follows. Assume the size of test suite T is n, and the number of statements

Algorithm Additional_FEP_Prioritization

Input: Test suite T, edge trace history TH_{edge}, FEP matrix M_{FEP}

Output: Prioritized test suite T'

```
1.  begin
2.      set T' empty
3.      initialize C(s), the confidence values for statements
4.      while T is not empty do
5.          for each test case t in T
6.              compute C_addi(t) according to current statements' confidence
7.          endfor
8.          select test case t that yields the greatest C_addi(t)
9.          append t to T'
10.         for each statement s covered by t
11.             update the confidence C(s)
12.         endfor
13.         remove t from T
14.     endwhile
15. end
```

FIGURE 3.6: Algorithm for additional FEP prioritization

in program P is m. The time required to calculate $C_{addi}(t)$ for a test t is $O(m \cdot QueryTime)$, where $QueryTime$ is the time required to query M_{FEP} to get $FEP(t, s)$. $QueryTime$ is a constant c once M_{FEP} is loaded. Thus, if T contains n test cases, the time required to select test t which yields the greatest $C_{addi}(t)$ is $O(n \cdot m)$. This cost dominates the cost of the while loop between

lines 4 and 14. The while loop itself executes n times. Therefore, the overall time complexity of the algorithm is $O(n^2 \cdot m)$.

Although the initial values of $C(s)$ can be set differently for different statements, we initialize all $C(s)$ to a fixed value for simplicity. The fixed value we choose for our experiment is 0, which means there is no confidence in any statement before running the test suite. (We may choose other initial values. For example, 0.5 can be used to indicate that the possibility of a statement being correct and containing a fault are equal.)

One difference between additional FEP prioritization and additional edge coverage prioritization is that in the additional FEP prioritization algorithm, we do not check whether "full confidence", which means that no additional confidence can be gained for all remaining tests, is achieved. The reason is that for a test t's $C_{addi}(t)$ to be 0, it requires all $C(s)$ of statements covered by t to be 1; and for a statement's confidence $C(s)$ to be increased to 1, there must be a test t whose $FEP(t, s)$ is 1; However, $FEP(t, s)$ is unlikely to be estimated as 1 in practice, because this means that test t can reveal any possible faults contained in s.

3.2.8 Total statement coverage prioritization

Total statement coverage prioritization is the same as total edge coverage prioritization, except that test coverage is measured in terms of program statements (nodes) rather than edges. We need to build the statement trace history of the test suite for the program.

3.2.9 Additional statement coverage prioritization

Similarly, additional statement coverage prioritization is the same as additional edge coverage prioritization, except that test coverage is measured in terms of program statements (nodes) rather than edges. In this algorithm, we also need a method to prioritize the remaining tests after complete statement coverage has been achieved by selected test cases. The method we use in this work is to assume that all statements are not yet covered, and apply the additional statement coverage prioritization to the remaining tests again.

Chapter 4

EMPIRICAL RESULTS

To further understand prioritization, and compare and evaluate prioritization techniques, we performed three experiments. In the following sections, we first describe the issues common to these experiments, then we discuss each experiment in detail.

4.1 Common issues

4.1.1 Research questions

In our empirical studies, we are investigating the following research questions:

Q1: Can test case prioritization techniques improve the rate of fault detection of test suites?

Q2: How do the various test case prioritization techniques presented in Chapter 3 compare to one another in terms of effects on rate of fault detection of test suites?

4.1.2 APFD measures

To investigate our research questions, we need to measure and compare the effect of using various prioritization techniques on rate of fault detection of test suites. *APFD*, a weighted average of the percentage of faults detected, is used

in our empirical study as a measurement of how rapidly a prioritized test suite detects faults. APFD values range from 0 to 1; higher APFD numbers mean faster (better) fault detection rates.

To illustrate the APFD measure, consider the following example: a program has 10 faulty versions and a test suite of 5 test cases, **A** through **E**. Table 4.1 shows the fault detecting ability of each of the 5 test cases.

Test Case	Fault									
	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

TABLE 4.1: Test suite and list of faults exposed.

Suppose we schedule test cases for execution in order **A–B–C–D–E** to form a prioritized test suite $\mathcal{X}$. Figure 4.1 (left) shows the percentage of *undetected* faults versus the fraction of the test suite $\mathcal{X}$ used. After running test case **A**, 2 of the 10 faults were detected; thus 80% of the faults remain undetected after $\frac{1}{5}$ of test suite $\mathcal{X}$ has been used. After running test case **B**, 2 more faults are detected and thus 60% of the faults remain undetected after $\frac{2}{5}$ of the test suite has been used. In Figure 4.1 (left), the area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults undetected over

the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the drop in the percentage of undetected faults. This interpolation is a granularity adjustment when only a small number of test cases comprise a test suite; the larger the test suite the smaller this adjustment.

Figure 4.1 (right) corresponds to Figure 4.1 (left) but shows the percentage of *detected* faults versus the fraction of the test suite used. The curve represents the cumulative percentage of faults detected. The shaded area under the curve represents the weighted average of the percentage of faults detected over the life of the test suite. This area is the prioritized test suite's average percentage faults detected measure; the APFD is 50% in this example.

Figure 4.2 reflects what happens when the order of test cases is changed to **E–D–C–B–A**. Let us call this prioritized test suite $\mathcal{Y}$. Figure 4.2 (left) clearly depicts that no faults remain undetected after $\frac{3}{5}$ of test suite $\mathcal{Y}$ has been used. This increase in the rate of detection is reflected in Figure 4.2 (right); the APFD over the entire suite has risen to 64%, indicating $\mathcal{Y}$ is "faster detecting" than $\mathcal{X}$.

Figure 4.3 shows the effects of using a prioritized test suite $\mathcal{Z}$ whose test case ordering is **C–E–B–A–D**. By inspection, it is clear that this ordering results in the earliest detection of the most faults and illustrates an optimal ordering. From Figure 4.3 (right) we see that the APFD of test suite $\mathcal{Z}$ is 84%, the best of the three.

The approach of using APFD to measure the effects of the prioritization techniques makes several assumptions. First, it assumes that all test cases have uniform cost. Second, it does not consider the difference between components

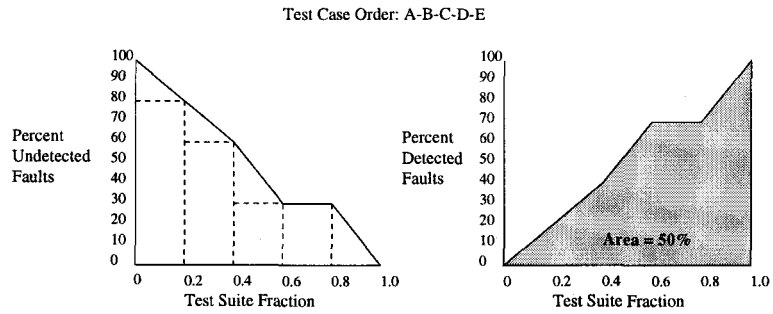

FIGURE 4.1: APFD for prioritized test suite $\mathcal{X}$: 50%.

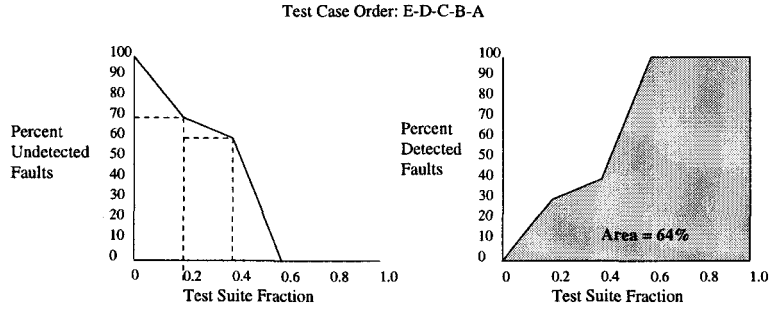

FIGURE 4.2: APFD for prioritized test suite $\mathcal{Y}$: 64%.

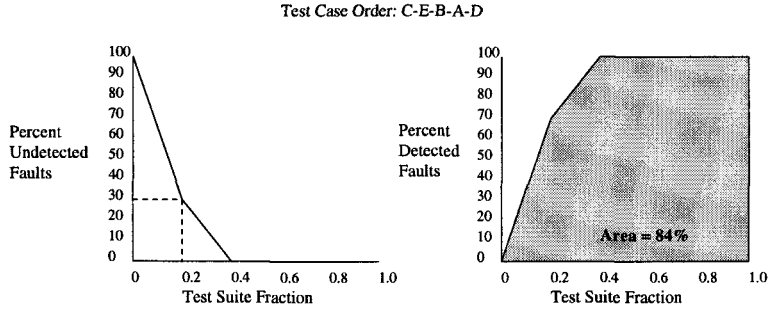

FIGURE 4.3: APFD for (optimal) prioritized test suite $\mathcal{Z}$: 84%.

Fault	1	2	3	4	5	6	7	8	9	10
$TF(f)$	1	3	3	3	1	2	2	5	5	5

TABLE 4.2: First detection time of test suite $\mathcal{X}$

of costs such as CPU time or human time. Third, all faults are equal; there are not some faults that are more important to be revealed than some other faults.

To calculate APFD, we need to define the *first detection time* of a fault. Given test suite T and program P, if f is a fault in P, and if in T, test cases $t_{i1}, t_{i2}, \cdots, t_{ik}$ reveal f, the first detection time TF of fault f is:

$$TF(f) = \min(i1, i2, \cdots, ik)$$

Table 4.2 shows the first detection time of the 10 faults of test suite $\mathcal{X}$.

4.1.3 Calculating APFD

We now give the formula for calculating the APFD values of test suites. In Figure 4.4 (left), given test suite $\mathcal{X}$, let S be the area below the curve. Then, the APFD of test suite $\mathcal{X}$ is $1 - S$. S contains two parts: S_1 is the area of the inscribed rectangles (dashed boxes); S_2 is the area of the triangles between S_1 and the curve.

To calculate S_1, let us first check Figure 4.4 (right). In Figure 4.4 (right), the vertical axis is the number of undetected faults, and the horizontal axis is the number of test cases used. Let S_1' be the area of the inscribed rectangles

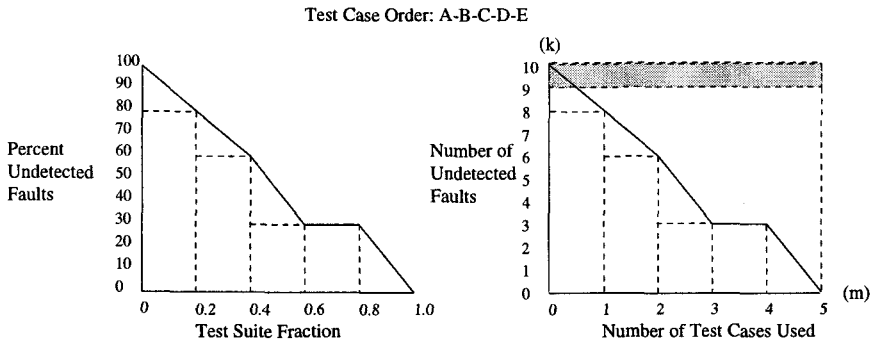

FIGURE 4.4: APFD calculation for prioritized test suite $\mathcal{X}$

(dashed boxes) in Figure 4.4 (right). Let m be the number of test cases of test suite $\mathcal{X}$, and let k be the number of faults revealed by $\mathcal{X}$, then,

$S_1 = \frac{S_1'}{m \cdot k}$

Assume the first detection times of these k faults are $TF(f_1)$, $TF(f_2)$, $\cdots$, $TF(f_k)$. The formula to calculate S_1' is:

$$S_1' = m \cdot k - [(m - TF(f_1) + 1) + (m - TF(f_2) + 1) + \cdots + (m - TF(f_k) + 1)] \tag{4.1}$$

In this formula, $m \cdot k$ represents the area of the whole rectangle. When f_1 is first revealed by test case at time $TF(f_1)$, from $TF(f_1)$ to m, the number of undetected faults should be decreased by 1. Therefore, a rectangle (the shaded dashed box) whose length is $m - TF(f_1) + 1$ and whose width is 1 should be deducted from the whole rectangle $(m \cdot k)$ in order to calculate S_1'. A similar analysis can be applied to other faults.

Simplifing Equation 4.1, we have:

$S_1' = TF(f_1) + TF(f_2) + \cdots + TF(f_k) - k$

Thus, the formula to calculate S_1 is:

$S_1 = \frac{TF(f_1)+TF(f_2)+\cdots+TF(f_k)}{m \cdot k} - \frac{1}{m}$

Now, consider S_2. If we merge all of the triangles together, we can see that their total area is a half of a rectangle whose length is 1 and width is $\frac{1}{m}$, so $S_2 = \frac{1}{2m}$. Combining all the above formulas together, we derive a formula for calculating the APFD value:

$$APFD = 1 - \frac{TF(f_1)+TF(f_2)+\cdots+TF(f_k)}{m \cdot k} + \frac{1}{2m}$$

4.1.4 Prioritization and analysis tools

To perform our empirical studies, we required several tools. Our test trace and control flow graph information was provided by the Aristotle program analysis system [6]. To obtain mutation scores for use in the FEP prioritization techniques we used the Proteum mutation system [3].

4.1.5 General environment and implementation

Our empirical studies were performed in a UNIX environment. The test case prioritization techniques outlined in Chapter 3 were implemented in GNU C. Meanwhile, we also created several other tools used in the experiments, including the tool to calculate APFD values of prioritized test suites, the tool to generate the FEP matrix (this matrix records the values of $FEP(t, s)$ for each test case and each statement, as described in Section 3.2.6) from the mutant data, and scripts to automate prioritization processes.

4.2 Experiment 1: Siemens programs

Our first experiment evaluated prioritization techniques on a set of small programs, and a fixed set of faults.

4.2.1 Subjects

The subject programs used in this experiment were seven C programs (see Table 4.3). Each program has a variety of versions, each containing one fault. Each program also has a large universe of test cases (test pool). These programs, versions, and inputs were assembled by researchers at Siemens Corporate Research for a study of the fault-detection capabilities of control-flow and data-flow coverage criteria [7]. We describe the other data in the table in the following paragraphs.

4.2.2 Faulty versions, test cases, and test suites

The researchers at Siemens created faulty versions of the seven base programs by manually seeding those programs with faults, usually by modifying a single line of code in the program. In a few cases they modified between two and five lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed the fault seeding, working "mostly without knowledge of each other's work" [7, p. 196].

For each base program, the researchers at Siemens created a large test pool containing possible test cases for the program. To populate these test pools, they first created an initial suite of black-box test cases "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code" [7, p. 194]. Then they add manually-created white-box test cases to the suite to ensure that each executable statement, edge, and definition-use pair in the base program or its control-flow graph was exercised by at least 30 test

cases. To obtain meaningful results with the seeded versions of the programs, the researchers retained only faults that were "neither too easy nor too hard to detect" [7, p. 196], which they defined as being detectable by at most 350 and at least 3 test cases in the test pool associated with each program.

Program	Lines of Code	No. of Versions	Test Pool Size	Test Suite Avg. Size
tcas	138	41	1608	6
schedule2	297	10	2710	8
schedule	299	9	2650	8
tot_info	346	23	1052	7
print_tokens	402	7	4130	16
print_tokens2	483	10	4115	12
replace	516	32	5542	19

TABLE 4.3: Siemens programs

To obtain sample test suites for these programs, we used the Siemens test pools and test-coverage information about the tests in those pools to generate 1000 edge-coverage-adequate test suites for each program. An edge-coverage-adequate test suite consists of test cases selected randomly from the test pool to achieve 100% coverage of coverable test cases. The random function we used is the C pseudo-random-number generator `rand`, seeded initially with the output of the C `times` system call.

More precisely, the algorithm to generate edge-coverage-adequate test suites is described in Figure 4.5. Table 4.3 lists the average sizes of the edge-coverage-adequate test suites generated by this algorithm for the Siemens programs.

Algorithm GenerateEdgeCoverageAdequateSuite
Input: Program P, test universe U, edge trace history TH_{edge}
Output: Edge-coverage-adequate test suite T

```
1. begin
2.       set T empty
3.       while an uncovered, coverable edge in program P remains
4.             randomly pick a test t from the universe U
5.             if t adds any new coverage then
6.                   append t to the test suite T
7.             endif
7.       endwhile
8. end
```

FIGURE 4.5: Algorithm for generating edge-coverage-adequate test suites

4.2.3 Experiment design

The experiment involves the following two independent variables:

- The subject program (7 programs, each with a variety of modified versions).

- The prioritization technique (unordered, random, optimal, edge-total, edge-addtl, FEP-total, FEP-addtl, stmt-total, stmt-addtl).

The experiment used a 7×9 factorial design with 1000 APFD measures per cell. That is, for each subject program P, we created 1000 edge-coverage-adequate test suites from its test pool. For each test suite, we then applied prioritization techniques $\mathcal{M}_2$ through $\mathcal{M}_9$, yielding 8 prioritized test suites. The original test suite (not reordered) was retained as a control; for analysis this was considered "prioritized" by technique $\mathcal{M}_1$. All together we created 63000 prioritized test suites. Then, their APFD values were evaluated and used as the statistical data set.

4.2.4 Data and analysis

Figure 4.6 depicts the effects of each prioritization technique on rates of fault detection of test suites. The boxplots[2] illustrate the APFD values of the 9 categories of prioritized test suites for each program and an all-program total. (Refer to Table 3.1 for a legend of the techniques.) $\mathcal{M}_1$ is the control group. $\mathcal{M}_2$ is the random prioritization group. $\mathcal{M}_3$ is the optimal prioritization group. Examining the boxplots of $\mathcal{M}_3$ with those of $\mathcal{M}_1$ and $\mathcal{M}_2$, it is apparent that optimal prioritization greatly improved the rate of fault detection (i.e., increased APFD values) of the test suites. Examining the boxplots of the other prioritization techniques, $\mathcal{M}_3$ through $\mathcal{M}_9$, it seems that all produce some improvement.

---

[2] A boxplot is a standard statistical device for representing a data set's distribution [8]. The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The middle of the three horizontal lines within the box represents the median. The vertical lines attached to the box indicate the tails of the distribution.

Using the SAS statistical package [4] to perform an ANOVA analysis, we can reject the null hypothesis that the APFD means for the various techniques were equal (α=.05), confirming our boxplot observations. However the ANOVA analysis indicated statistically significant cross-factor interactions: various programs *have* various effects on APFD values of prioritization techiniques. Thus general statements about prioritization technique effects must be qualified.

While rejection of the null hypothesis tells us that some techniques produce statistically different APFD means, to determine which techniques differ from each other requires running a multiple-comparison procedure [12]. Of the commonly used means separation tests, we elected to use the Bonferroni method — for its conservatism and generality.

Using Bonferroni, the minimum statistically significant difference between APFD means was calculated for each program. These are given in Table 4.4. The techniques are listed within each program subtable by their APFD mean values, from higher (better) to lower (worse). Grouping letters partition the techniques; techniques that are not significantly different share the same grouping letter.

Examining these sub-tables affirm what the boxplots indicated: that although the relative improvement provided by each technique is dependent on the program, all the heuristic techniques provided some significant improvement in rate of fault detection and in only one case, on `schedule`, did any heuristic not outperform the untreated or randomly prioritized test suites. Overall in our study, additional FEP prioritization outperformed all prioritization techniques based on coverage. Furthermore, total FEP prioritization outperformed all coverage-based techniques other than total edge coverage prioritization. However, these results did vary across individual programs and, where FEP-based

techniques did outperform coverage-based techniques, the total gain in APFD was not great. These results run contrary to our initial intuitions and suggest that given their expense, FEP-based prioritization may not be as cost-effective as coverage-based techniques.

Again considering overall results, it is interesting that total edge coverage prioritization outperforms additional edge coverage prioritization and that total statement coverage prioritization outperforms additional statement coverage prioritization. These effects, too, vary across the individual programs. Nevertheless, the worst-case costs of total edge and statement coverage prioritization are much less than the worst-case costs of additional edge and statement coverage prioritization; this suggests that the less expensive total-coverage prioritization schemes may be more cost-effective than additional-coverage schemes. This result, too, runs counter to our intuitions.

Another effect worth noting is that on five of the seven programs, randomly prioritized test suites outperformed untreated test suites. We conjecture that this difference is due to the type of test suites and faults used in the study. As described in Section 4.2.2, our test suites were generated for coverage by greedily selecting tests from test pools; the order in which tests were added to suites during this process constitutes their untreated order. We suspect that this process caused test cases added to the "ends" of the test suites to cover (on average) harder to reach statements than test cases added to the "beginnings" of the test suites. The faults embedded in the Siemens programs are relatively hard to detect; a disproportionate number reside in harder-to-reach statements and are detected (on average) by test cases that are added later to the test suites. Random prioritization essentially redistributes test cases that reach and

expose these faults throughout the test suites, causing the faults to be detected more quickly.

4.2.5 Threats to validity

In this section, we discuss some of the potential threats to the validity of this experiment. The primary ones are threats to external validity that limit our ability to generalize our results. Our primary concern involves the representativeness of the artifacts utilized. First, the subject programs, though nontrivial, are small and larger programs may be subject to different cost-benefit tradeoffs. Also, there is exactly one seeded fault in every subject program; in practice, programs have much more complex error patterns. Finally, the test suites we utilized represent only one type of test suite that could appear in practice.

4.3 Experiment 2: Siemens programs with greater fault base

To begin to address the threats to validity for study 1, we next performed an experiment using a wider range of faulty versions.

4.3.1 Subjects

The subject programs used in the experiment are also the seven Siemens programs, as in our first experiment. However, the faulty versions of these programs are different.

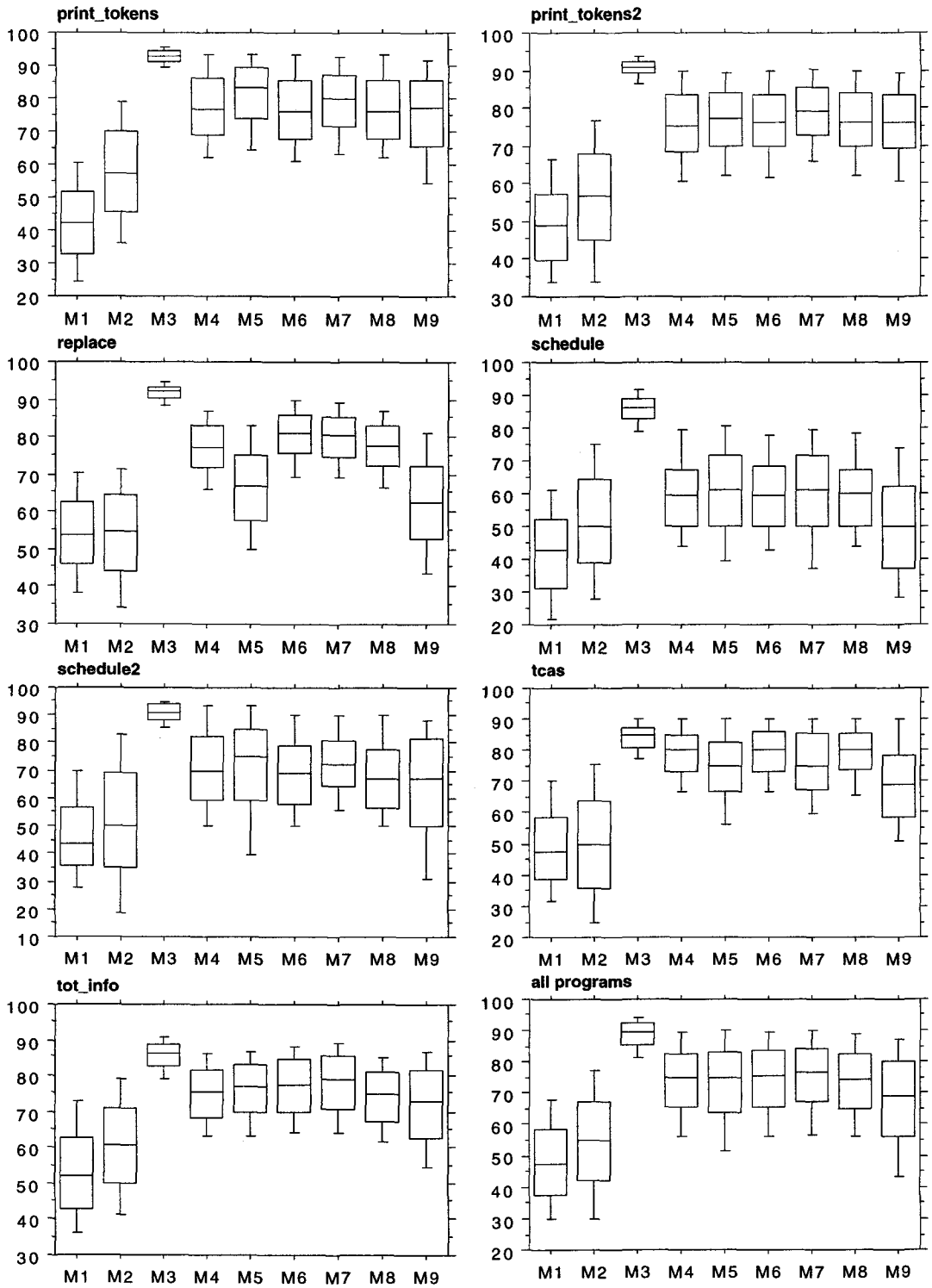

FIGURE 4.6: APFD boxplots for experiment 1

print_tokens

Grouping	Mean	Technique
A	92.5461	optimal
B	80.8842	edge-addtl
C	78.2727	FEP-addtl
D C	76.8573	edge-total
D E	76.4770	FEP-total
D E	76.4647	stmt-total
E	74.8199	stmt-addtl
F	57.2829	random
G	42.6163	untreated

df= 8991 MSE= 155.0369 Critical Value of T= 3.20
Minimum Significant Difference= 1.7808 (α=.05)

print_tokens2

Grouping	Mean	Technique
A	90.5152	optimal
B	78.3211	FEP-addtl
C	76.1678	edge-addtl
C	75.8848	stmt-total
C	75.7985	FEP-total
C	75.5995	stmt-addtl
C	74.8830	edge-total
D	55.9729	random
E	49.3272	untreated

df= 8991 MSE= 124.203 Critical Value of T= 3.20
Minimum Significant Difference= 1.5939 (α=.05)

replace

Grouping	Mean	Technique
A	91.6901	optimal
B	80.0171	FEP-total
B	79.6959	FEP-addtl
C	77.1355	stmt-total
C	76.8482	edge-total
D	66.5639	edge-addtl
E	62.3795	stmt-addtl
F	54.4460	untreated
F	54.0668	random

df= 8991 MSE= 110.782 Critical Value of T= 3.20
Minimum Significant Difference= 1.5053 (α=.05)

schedule

Grouping	Mean	Technique
A	85.7074	optimal
B	60.6765	edge-addtl
B	59.8694	stmt-total
B	59.8484	FEP-addtl
B	59.6161	edge-total
B	59.4430	FEP-total
C	51.4087	random
C	50.4418	stmt-addtl
D	41.9670	untreated

df= 8991 MSE= 222.3662 Critical Value of T= 3.20
Minimum Significant Difference= 2.1327 (α=.05)

schedule2

Grouping	Mean	Technique
A	90.1794	optimal
B	72.0518	FEP-addtl
B	70.6432	edge-total
C B	70.2513	edge-addtl
C D	68.0438	FEP-total
D	67.5409	stmt-total
E	63.7391	stmt-addtl
F	51.3077	random
G	47.0302	untreated

df= 8127 MSE= 280.635 Critical Value of T= 3.20
Minimum Significant Difference= 2.5199 (α=.05)

tcas

Grouping	Mean	Technique
A	83.8845	optimal
B	78.9253	stmt-total
B	78.7998	FEP-total
B	78.5781	edge-total
C	75.1880	FEP-addtl
D	73.3552	edge-addtl
E	68.5357	stmt-addtl
F	50.1038	random
F	49.4311	untreated

df= 8973 MSE= 148.5302 Critical Value of T= 3.20
Minimum Significant Difference= 1.7447 (α=.05)

tot_info

Grouping	Mean	Technique
A	85.4258	optimal
B	77.5442	FEP-addtl
C B	76.8218	FEP-total
C D	75.8798	edge-addtl
E D	74.8807	edge-total
E	73.9979	stmt-total
F	71.4503	stmt-addtl
G	60.0587	random
H	53.1124	untreated

df= 8991 MSE= 110.4918 Critical Value of T= 3.20
Minimum Significant Difference= 1.5033 (α=.05)

All Programs

Grouping	Mean	Technique
A	88.5430	optimal
B	74.4501	FEP-addtl
C	73.7049	FEP-total
D C	73.2205	edge-total
D	72.9030	stmt-total
E	71.9919	edge-addtl
F	66.7502	stmt-addtl
G	54.3575	random
H	48.2927	untreated

df= 62055 MSE= 162.9666 Critical Value of T= 3.20
Minimum Significant Difference= 0.6948 (α=.05)

TABLE 4.4: Bonferroni means separation tests for experiment 1

4.3.2 Faulty versions, test cases, and test suites

In our first experiment, to perform FEP-based prioritization, we created mutants for each base program and obtained mutation scores of the tests in order to build FEP matrixes. These mutants, which are copies of the original programs into which small modifications are seeded, can be utilized as faulty versions of the base programs [3]. As we described in Section 4.2.5, one of the threats to external validity of our empirical study is that the faulty versions we used are not sufficient to represent the range of error patterns that occur in practice. Therefore, in this experiment, we let these mutants be the faulty versions of the base programs, and investigated the effects of test case prioritization techniques on the rate of detecting these faults (mutants).

Table 4.5 shows the number of mutants of each base program.

The test pools and test suites used for the base programs are the same as those used in the first experiment.

4.3.3 Experiment design

Since the subject programs and their test suites were kept the same, we did not need to generate prioritized test suites again. However, for each base program,

---

[3] One issue with program mutation involves determining semantic equivalence between the mutant and the original version. A semantically equivalent mutant can never be killed by any tests. Determining semantic equivalence is difficult, and was not feasible to accomplish in this study, given the number of mutants involved. Therefore, one approach is to consider mutants never killed by any tests in the test pools as semantically equivalent mutants. This approach, however, may overestimate the number of semantically equivalent mutants, for even though there is a mutant has not been killed by any tests used in our mutation analysis, we can not guarantee that it is semantically equivalent. A second approach is to ignore the possibility of having semantically equivalent mutants. Of course, this approach underestimates the number of semantically equivalent mutants. Thus, the first approach may cause us to overestimate statement sensitivity, and the latter to underestimate it. We choose the second one due to its conservatism.

Program	totinfo	schedule1	schedule2	tcas	printtok1	printtok2	replace
Mutants	5898	2153	2828	2876	4030	4346	9622

TABLE 4.5: Number of mutants of Siemens programs

we had to rebuild its fault detection matrix from the mutant data, and then evaluate the APFD values of these 63000 test suites again.

4.3.4 Data and analysis

Figure 4.7 presents boxplots of the APFD values of the 9 categories of prioritized test suites for each program and an all-program total. It is similar to Figure 4.6 but APFD values are calculated based on different faulty versions for each base program. Table 4.6 gives results of Bonferroni means separation tests.

Examining Figure 4.7 and Table 4.6, prioritization techniques $\mathcal{M}_3$ through $\mathcal{M}_9$ produce improvements in APFD values of test suites. Similar to experiment 1, considering overall results, additional FEP prioritization outperformed all other prioritization techniques. Furthermore, total FEP prioritization outperformed all coverage-based techniques. However, as in experiment 1, these results did vary across individual programs and, where FEP-based techniques did outperform coverage-based techniques, the total gain in APFD was not great.

Overall in experiment 2, additional edge coverage prioritization outperformed total edge coverage prioritization but total statement coverage prioritization outperformed additional statement coverage prioritization. These effects, too, vary across the individual programs. This further suggests that

additional-coverage prioritization may not be as cost-effective as total-coverage prioritization.

The range of faults used in this experiment is wider than the range of faults used in the first experiment. We have both faults which are easy to detect and faults which are relatively hard to detect. In this experiment, randomly prioritized test suites did not outperform untreated test suites. This confirms our conjecture about the reason that randomly prioritized test suites have better APFD values than untreated test suites presented in Section 4.2.4.

4.4 Experiment 3: Space program

One threat that limits our ability to generalize the results of our first two experiments is that in neither experiment are the faults we used faults that occured in practice. In this experiment, we used as our subject a real program that contains real faults found during its testing stage.

4.4.1 Subjects

The subject used in this experiment is a program developed for the European Space Agency. Its purpose is to allow users to describe the configuration of an array of antennas using array definition language (ADL). The space program first reads an ADL file which consists of several ADL statements, then checks the grammer and consistency rules for these ADL statements. If the ADL file contains errors, the program reports the corresponding error messsage to the users. If the ADL file is correct, the program outputs a data file of a complete list of elements, positions and excitations that describe the configuration of

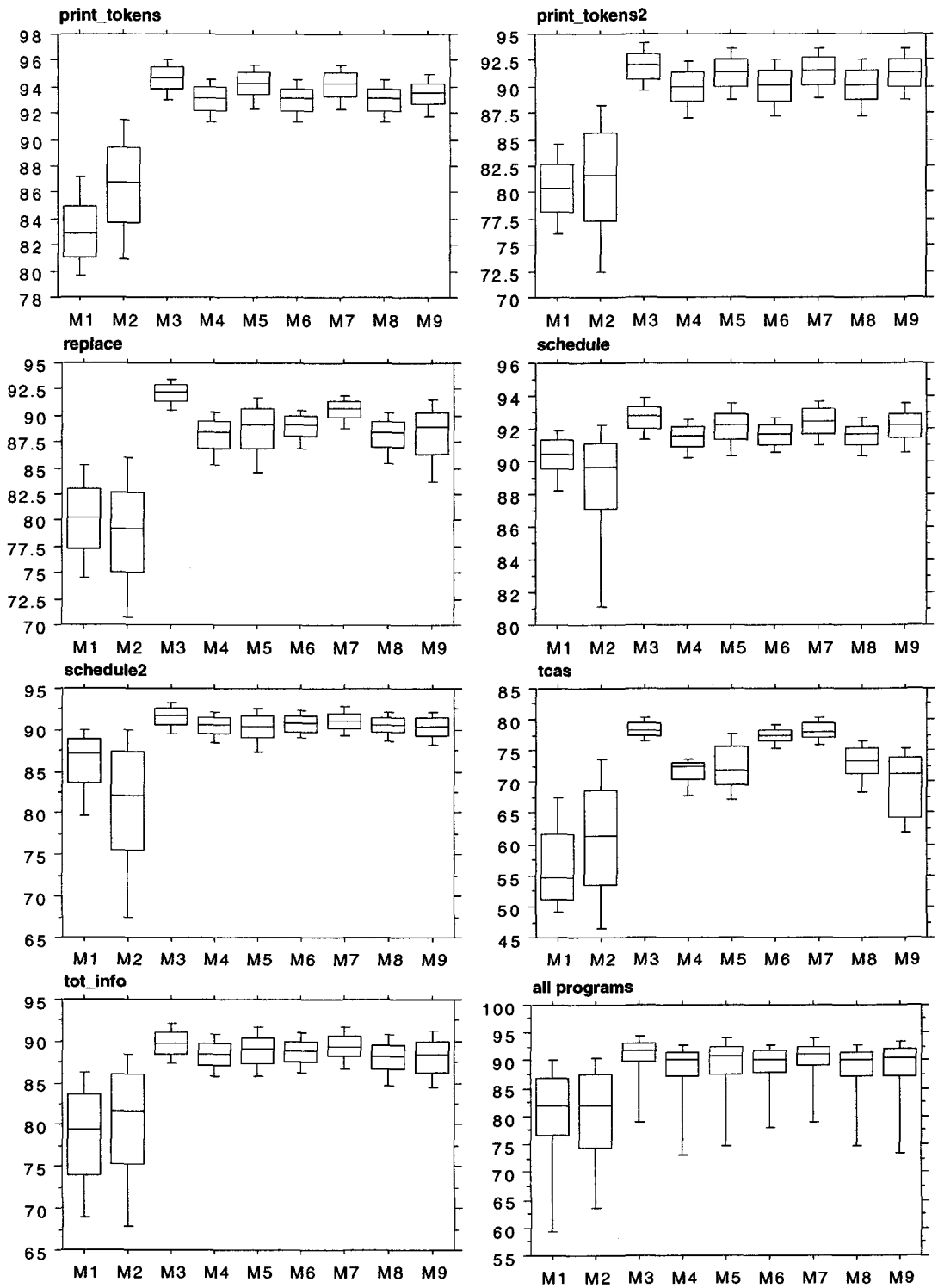

FIGURE 4.7: APFD boxplots for experiment 2

print_tokens		
Grouping	Mean	Technique
A	94.5726	optimal
B	94.1552	edge-addtl
B	94.0983	FEP-addtl
C	93.4522	stmt-addtl
D	92.9773	edge-total
D	92.9769	FEP-total
D	92.9676	stmt-total
E	86.3318	random
F	83.0611	untreated
df= 8991 MSE= 4.08951 Critical Value of T= 3.20		
Minimum Significant Difference= 0.2892 (α=.05)		

print_tokens2		
Grouping	Mean	Technique
A	91.8595	optimal
B	91.3132	FEP-addtl
B	91.2169	edge-addtl
B	91.1878	stmt-addtl
C	89.9208	stmt-total
C	89.9189	FEP-total
C	89.8104	edge-total
D	80.9336	random
E	80.3819	untreated
df= 8991 MSE= 8.62951 Critical Value of T= 3.20		
Minimum Significant Difference= 0.4201 (α=.05)		

replace		
Grouping	Mean	Technique
A	92.0247	optimal
B	90.4726	FEP-addtl
C	88.8712	FEP-total
D C	88.5446	edge-addtl
D E	88.1150	stmt-addtl
E	88.1000	stmt-total
E	88.0152	edge-total
F	80.0455	untreated
F	78.6597	random
df= 8991 MSE= 9.03445 Critical Value of T= 3.20		
Minimum Significant Difference= 0.4299 (α=.05)		

schedule		
Grouping	Mean	Technique
A	92.6215	optimal
B A	92.3722	FEP-addtl
B C	92.1101	stmt-addtl
C	92.0307	edge-addtl
D	91.6136	FEP-total
D	91.5112	stmt-total
D	91.4804	edge-total
E	90.0178	untreated
F	88.2226	random
df= 8991 MSE= 3.91423 Critical Value of T= 3.20		
Minimum Significant Difference= 0.283 (α=.05)		

schedule2		
Grouping	Mean	Technique
A	91.4701	optimal
B A	90.9788	FEP-addtl
B C	90.7122	FEP-total
C D	90.5303	stmt-total
C D	90.3959	edge-total
C D	90.2108	stmt-addtl
D	90.0740	edge-addtl
E	85.7386	untreated
F	80.3034	random
df= 8991 MSE= 12.9149 Critical Value of T= 3.20		
Minimum Significant Difference= 0.514 (α=.05)		

tcas		
Grouping	Mean	Technique
A	78.3102	optimal
A	78.1003	FEP-addtl
B	77.2512	FEP-total
C	72.9459	stmt-total
C	72.2930	edge-addtl
D	71.5644	edge-total
E	69.2863	stmt-addtl
F	60.6226	random
G	56.7793	untreated
df= 8991 MSE= 24.0915 Critical Value of T= 3.20		
Minimum Significant Difference= 0.702 (α=.05)		

tot_info		
Grouping	Mean	Technique
A	89.6274	optimal
B A	89.2970	FEP-addtl
B C	88.7839	edge-addtl
C	88.6767	FEP-total
D C	88.3099	edge-total
D	87.9411	stmt-addtl
D	87.9338	stmt-total
E	79.8922	random
F	78.1495	untreated
df= 8991 MSE= 15.7992 Critical Value of T= 3.20		
Minimum Significant Difference= 0.56851 (α=.05)		

All Programs		
Grouping	Mean	Technique
A	90.0694	optimal
B	89.5189	FEP-addtl
C	88.5744	FEP-total
D	88.1569	edge-addtl
E	87.7013	stmt-total
F	87.5076	edge-total
F	87.4719	stmt-addtl
G	79.2808	random
G	79.1676	untreated
df= 62055 MSE= 11.2104 Critical Value of T= 3.20		
Minimum Significant Difference= 0.1809 (α=.05)		

TABLE 4.6: Bonferroni means separation tests for experiment 2

the antenna array. The space program consists of three subsystems: parser, computation, and formatting. It consists of about 6288 lines of C code.

4.4.2 Faulty versions, test cases, and test suites

Initially, 33 faults of the space program had been revealed during its testing and integration phases; these faults were provided to us. When we designed new test cases for the space program, we found another five faults. Thus, altogether we have 38 faulty versions for the space program. Among these 38 faulty versions, some versions are special: versions 1, 2 and 32 are actually semantically equivalent versions of the base program. No test case can cause them to behave differently from the base program. Therefore, we believe that these three versions could not yield meaningful results for our empirical study, and we eliminated them from consideration.

At first, we were provided with 10000 test cases for the space program. These test cases had been generated randomly by Frankl and Vokolos for use in an earlier study [19]. However, these tests did not cover all the code; thus, we added new test cases to let all reachable nodes and edges in the control flow graph of the base program be covered by at least 30 test cases. This yielded a test pool of 13585 test cases.

Using this test pool, we built 1000 edge-coverage-adequate test suites by applying the algorithm described in Figure 4.5.

4.4.3 Experiment design

Since the time required to perform mutation analysis for all tests of the space program is very long (we have 132163 mutants), we could not use all 1000 test

suites. Instead, we analyzed the results produced by test cases in 50 randomly selected test suites, and applied our FEP-based prioritizations to these 50 suites.

For the other prioritization techniques, we used all 1000 test suites.

4.4.4 Data and analysis

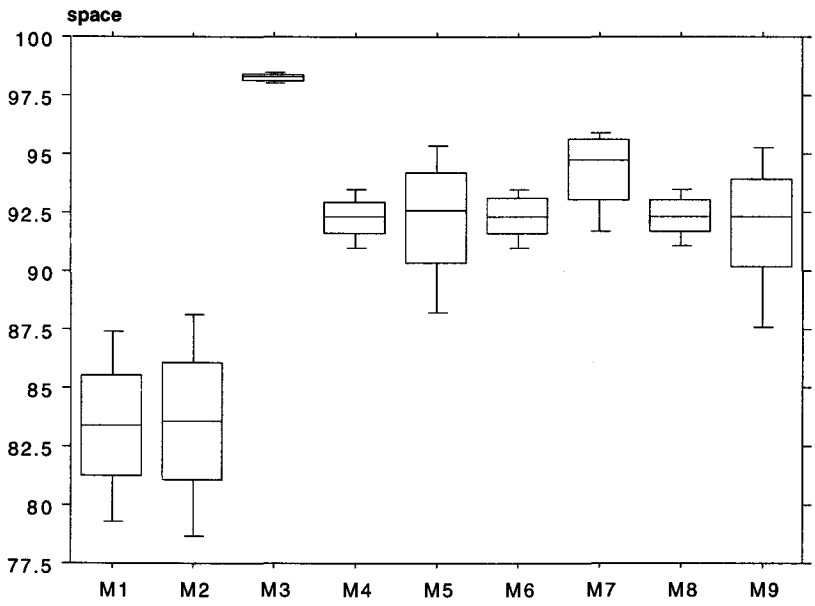

FIGURE 4.8: APFD boxplot for experiment 3

Figure 4.8 presents the boxplot of the APFD values for the 9 categories of prioritized test suites for the space program. Similarly, we analyzed the minimum statistically significant difference between APFD means for the space program. The results are given in Table 4.7.

space		
Grouping	Mean	Technique
A	98.2718	optimal
B	94.2316	FEP-addtl
C	92.3115	stmt-total
C	92.2944	FEP-total
C	92.2564	edge-total
C	92.0828	edge-addtl
C	91.8104	stmt-addtl
D	83.4450	random
D	83.3274	untreated
df= 7091 MSE= 6.0074 Critical Value of T= 3.20		
Minimum Significant Difference= 0.8011 (α=.05)		

TABLE 4.7: Bonferroni means separation tests for experiment 3

Examining Figure 4.8 and Table 4.7, prioritization techniques $\mathcal{M}_3$ through $\mathcal{M}_9$ produce improvements in APFD values of test suites. Among them, additional FEP prioritization stands out as different and better than the other techniques. There was no significant difference, however, among the four coverage-based techniques and total FEP prioritization. This again suggests that the less expensive total-coverage prioritization may be more cost-effective than additional-coverage prioritization. Also for the space program, and similar to the results of experiment 2, randomly prioritized test suites and untreated test suites are indistinguishable.

Chapter 5

CONCLUSION AND FUTURE WORK

In this paper, we have described several test case prioritization techniques, and empirical studies performed to investigate their relative abilities to improve how quickly faults can be detected by test suites. The results of our studies indicate that test case prioritization can substantially improve the rate of fault detection of test suites, and that this result occurs even for the least sophisticated (and hence least expensive) techniques.

The results of our study suggest several promising areas for further exploration.

First, additional studies utilizing other programs and types of test suites, as well as a wider range and distribution of faults, are necessary in order to increase our ability to generalize our empirical results.

Second, alternative measures of prioritization effectiveness may be possible.

Third, it may be fruitful to investigate the relationship between the effectiveness of various prioritization techniques and type of faults. For example, tester may wish to increase the rate of detection of some high-risk faults.

Fourth, because our analysis revealed a sizeable performance gap between prioritization heuristics and greedy-optimal prioritization, and our FEP-based techniques did not bridge this gap, alternative techniques that improve upon the FEP approaches would be useful. Techniques that incorporate static measures of fault-proneness [10] may also be of interest.

Fifth, our studies showed that given their expense, FEP-based prioritization techniques may not be as cost-effective as coverage-based techniques. The studies also suggested that less expensive total-coverage prioritization may be more cost-effective than additional-coverage prioritization. These arguments, however, do not measure precisely the cost of prioritization techniques. One cost that can be considered with respect to prioritization techniques is the time required to execute prioritization tools. However, if prioritization tools can be run during off-peak hours, this cost may be noncritical. It may be useful to build more precise models to measure the cost-effectiveness of prioritization techniques under certain circumstances.

Sixth, there are various goals that can be addressed by test case prioritization. One alternative goal is to improve the rate of increasing the reliability of a system. It is worth exploring further the effects of prioritization techniques on meeting other goals.

Finally, the test case prioritization problem, in general, has many more facets than we have here considered. The test case prioritization techniques that we have examined can be described as "general prioritization techniques" in the sense that they are applied to a base version of a program, with no knowledge of the location (or probable location) of modifications to the software, in the hopes of producing a test case ordering that will be effective over subsequent (and as yet unknown) versions of the software. Such general techniques could also incorporate information on probabilities of modification. Alternative techniques could utilize knowledge of the location of modifications to prioritize test cases for a particular modified version.

Through the results reported in this paper, and this future work, we hope to provide software practitioners with useful, cost-effective techniques for improving regression testing processes through prioritization of test cases.

BIBLIOGRAPHY

[1] T. Ball. On the limit of control flow analysis for regression test selection. In *ACM Int'l Symp. on Softw. Testing and Analysis*, pages 134–142, March 1998.

[2] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 211–222, May 1994.

[3] Márcio E. Delamaro and José C. Maldonado. Proteum–A tool for the assessment of test adequacy for C programs. In *Proc. of the Conf. on Performability in Computing Sys. (PCS 96)*, pages 79–95, New Brunswick, NJ, July 25–26 1996.

[4] Rudolf J. Freund and Ramon C. Littell. *SAS for Linear Models: A guide to the ANOVA and GLM procedures.* SAS Institute Inc., Cary, NC, 1981.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York, 1979.

[6] M.J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar 1997.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[8] R. Johnson. *Elementary Statistics.* Duxbury Press, Belmont, CA, sixth edition, 1992.

[9] H.K.N. Leung and L. White. Insights into regression testing. In *Proc. of the Conf. on Softw. Maint.*, pages 60–69, October 1989.

[10] J.C. Munson and T.M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. on Softw. Eng.*, pages 423–433, May 1992.

[11] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. of the ACM*, 41(5):81–86, May 1988.

[12] Lyman Ott. *An Introduction to Statistical Methods and Data Analysis.* PWS-Kent Publishing Company, Boston, MA, third edition, 1988.

[13] R. Pressman. *Softw. Eng.: A Practitioner's Approach.* McGraw-Hill, New York, NY, 1987.

[14] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. on Softw. Eng.*, 22(8):529–551, August 1996.

[15] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, April 1997.

[16] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. of Conf. on Softw. Maint.*, pages 34–43, November 1998.

[17] J. Voas. PIE: A dynamic failure-based technique. *IEEE Trans. on Softw. Eng.*, pages 717–727, August 1992.

[18] F.I. Vokolos and P.G. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Proc. of the 3rd Int'l. Conf. on Rel., Quality & Safety of Softw.-Intensive Sys. (ENCRESS '97)*, May 1997.

[19] F.I. Vokolos and P.G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proc. of Conf. on Softw. Maint.*, pages 44–53, November 1998.

[20] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th Int'l. Conf. on Softw. Eng.*, pages 41–50, April 1995.

[21] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proc. of the Eighth Intl. Symp. on Softw. Rel. Engr.*, pages 230–238, November 1997.